Tabula Rasa: Starting Safe Stays Safe

Tyler Potyondy
tpotyondy@ucsd.edu
UC San Diego

Samir Rashid
s3rashid@ucsd.edu
UC San Diego

Leon Schuermann
lschuermann@princeton.edu
Princeton University

Anthony Tarbinian
atarbini@ucsd.edu
UC San Diego

Pat Pannuto
ppannuto@ucsd.edu
UC San Diego

Abstract

Low power devices, sensors, and real time systems are increasingly connected. With this connectivity, embedded devices now face a more complex attack surface, underscoring the importance of device security. Embedded operating systems are able to span diverse application and hardware domains because they are highly configurable. This flexibility, however, implies that downstream embedded applications may be *flexible* in how they use security features. This paper investigates how downstream applications use configurable security features in practice. We find that a majority of applications do not alter the default configuration provided by their chosen runtime, and as a result, do not utilize available security options. Early evidence suggests that this under-utilization is due to both runtime and development overhead.

CCS Concepts

• **Software and its engineering** → *Software design tradeoffs*; • **Security and privacy** → **Operating systems security**.

Keywords

Embedded OS, Security, IoT

1 Introduction

Embedded and connected devices have become a ubiquitous and integral part of our world. From medical devices and smart home products to industrial controllers, these devices perform an increasing set of complex, critical tasks. To abstract these complexities, embedded operating systems (OSes) evolved to provide rich development environments and high-level abstractions, such as platform-independent hardware models and separable tasks. Despite the availability of these abstractions and isolation mechanisms for tasks, we find that in practice many embedded applications lack strong isolation and do not enable security reinforcements. This paper explores projects using embedded OSes to quantify how configurable security features are used and how they factor into the given project's design.

We focus on microcontroller-class embedded devices, i.e. those that are severely resource constrained in terms of memory, compute, and, often, energy. These systems typically have 64 kB or less of SRAM, have simple in-order cores running at tens of MHz, and may not have access to constant power. As a result, embedded OSes must differ significantly from traditional OSes in how they provide feature-rich libraries, concurrency, and security isolation. For instance, embedded OSes are often aggressively modular to provide flexibility as unused features can adversely affect binary sizes and runtime overhead. In many cases, even isolation and security protections can be and are relaxed in favor of performance. Given the modular and configurable nature of embedded operating systems, we ask the question: *do applications built upon embedded OSes enable configurable, opt-in security features?* This paper studies popular, open source embedded OSes to answer this question and gather insights around end-application usage of configurable and opt-in security features.

Historically, embedded devices did not offer connectivity. For instance, industrial controllers or automotive Electronic Control Units (ECUs) required a technician physically access and "plug-into" the device. As embedded devices have evolved into the Internet of Things (IoT), many now offer internet connectivity to gather data, offer remote control, and provide firmware updates. Developers now expect embedded OSes to provide library features such as over-the-air updates and network stacks. Internet connectivity and the global scale of many billions of deployed devices implies that embedded devices now face an attack surface more akin to that of a traditional operating system. As such, the security embedded OSes provide is paramount.

Manufacturers of embedded devices have reacted to these risks, and the need for secure systems, by introducing isolation mechanisms such as memory protection units (MPUs) and ARM TrustZone [4]. Some proprietary OSes, such as SafeRTOS, require the usage of MPUs to isolate processes[1] and guard against stack overflows. This provides enhanced security for safety critical applications [10]. Many general purpose embedded OSes, however, leave decisions around MPU-based isolation up to downstream developers. Subsequently, the responsibility to integrate these mechanisms and secure such applications falls upon the developer.

To investigate the usage of configurable security features, we first provide background on the OSes we study. Following this, Section 3 presents our survey of repositories on GitHub, in addition to a quantitative analysis of memory protection overheads. In Section 4, we further provide case studies that analyze two particular applications in more detail. Finally, based on these results and studies, Section 5 proposes likely causes for applications' under-utilization of configurable security features.

2 Background

This study analyzes three popular, open-source embedded OSes: FreeRTOS, Zephyr, and RIOT. These systems were selected to provide a sampling of projects varying in popularity, longevity, and

[1]We consider the terms tasks and processes interchangeable in this paper.

	Hardware Stack Guard	Kernel/Process Isolation	Process/Process Isolation	MPU Support
FreeRTOS	○	○	○	○
RIOT	●	×	×	●
Zephyr	○	○	○	○

× No support ○ Supported, default-off ● Supported, default-on

Table 1: Survey of features available across embedded OSes.

target focus. FreeRTOS and Zephyr represent two popular projects with widespread adoption. RIOT is geared towards usability in IoT.

In this section, we provide a brief overview of the OSes we study. We consider their primary advertised use case(s), configuration parameters, and whether security features are enabled by default. We do not consider OSes lacking MPU support, such as Contiki-NG [8]. Moreover, we exclude OSes such as SafeRTOS [10] or TockOS [5] that mandate these features, as we are interested in what downstream developers do when presented with a choice. Prior work investigates embedded operating systems' support for memory protection and isolation primitives [13]. In this work, Zhou et al. survey embedded OS support for code integrity protection, data execution prevention, coarse-grained stack guard, kernel memory isolation, user task memory isolation, and peripherals isolation. We select a subset of this taxonomy to investigate *in the wild*, namely MPU-based stack protection and process isolation.[2] Our chosen taxonomy is shown in Table 1. This subset aligns with the configurable security features available for the particular OSes we study.

2.1 FreeRTOS

FreeRTOS is a real time embedded operating system for microcontrollers that offers support for many core libraries and Amazon Web Services IoT integrations [1].

Applications built upon FreeRTOS specify kernel configurations using `#define` statements in the `FreeRTOSConfig.h` file. FreeRTOS provides MPU support for a number of ARMv7-M and ARMv8-M microcontrollers [2]. By default, all FreeRTOS processes are considered privileged and are spawned with the `xTaskCreate` function call. Privileged processes grant unrestricted access to peripherals and memory. FreeRTOS supports process isolation and stack overflow protection using the `xTaskCreateRestricted` function call. Subsequently, developers using FreeRTOS configure their usage of MPU security by deciding whether to use privileged or restricted processes.

FreeRTOS also offers software stack overflow detection that can be enabled with the `configCHECK_FOR_STACK_OVERFLOW` flag. FreeRTOS software stack guards serve as wrappers to confirm the stack is valid following a context switch. Notably, software stack guards are added automatically during compilation (if enabled by the developer) and "introduce a context switch overhead" [3]. FreeRTOS is the only surveyed OS that provides support for software stack guards. Given that software stack guard's impose a performance but not developer overhead (i.e. automatically compiled if enabled), surveying their usage provides insight to configurable feature usage in light of developer and performance overhead.

[2]We define process isolation for the purpose of this paper to include both process-process and kernel-process isolation.

2.2 Zephyr

Zephyr is a real time embedded OS, hosted under the Linux Foundation, that 'provides cross platform support for over 600 boards and emphasizes portability and connectivity' [12]. Zephyr provides support for MPUs, hardware stack protection, and process isolation that must be enabled explicitly [11]. Configuration of these constants occurs in a project-defined `.conf` file. To enable usermode processes, Zephyr developers must enable the `CONFIG_USERSPACE` flag. Notably, Zephyr usermode processes run at a reduced privilege level, may only access the stacks of processes within the same memory domain, and, by default, will detect process stack overflows. To detect stack overflows in privileged processes, the developer must enable the `CONFIG_HW_STACK_PROTECTION` flag.

2.3 RIOT

RIOT advertises itself as the "*friendly Operating System for IoT*" [9]. This embedded OS offers a preemptive, tickless scheduler, support for over 200 boards, and extensive network stack support. RIOT offers hardware stack protection by default on all boards that feature an ARMv7-M MPU. Developers can opt-out of this protection feature by disabling the `CONFIG_MPU_STACK_GUARD` flag. RIOT does not support process isolation.

3 Methodology & Results

To quantify application usage of configurable embedded OS security parameters, we conduct a survey of GitHub projects using the selected embedded OSes. We also benchmark Zephyr's MPU-imposed runtime overhead. In this section, we present our methodology and results for the survey and benchmarking.

3.1 Configurable Security Analysis

We first create a dataset of applications built using each embedded OS. To form this dataset, we employ the following process:

(1) Search GitHub to collect repositories whose metadata contains the name of the operating system or contains a code snippet/filename unique to that operating system.
(2) Union these two searches, excluding forks.

The GitHub API search queries are shown in Table 2. One limitation of our repository search is that GitHub limits each search to 1000 results. The metadata search uses the GitHub *repository search* API which allows for results to be sorted by stars. For the purposes of our study, this helps to ensure the 1000 repositories obtained using the *repository search* are the most popular repositories. In addition to the *repository search*, we also utilize GitHub's *code search* API to gather projects with repository metadata insufficient to be found by the *repository search*.

Our survey faces the challenge of potentially being skewed towards "toy projects" that are perhaps underdeveloped. We mitigate this risk by excluding stale projects (i.e. most recent commits older than five years) and projects with less than 10 stars. Although stars are an imperfect measure of a repository's use, this serves as a proxy

Embedded OS	GitHub Query Terms	Search Type
FreeRTOS	`freertos`	**(RS)**
	`filename:FreeRTOSConfig.h`	**(CS)**
Zephyr	`zephyr`	**(RS)**
	`filename:west.yml`	**(CS)**
RIOT	`RIOTBASE ?=`	**(CS)**

Table 2: GitHub query terms used for each project's repository search (RS) and code search (CS).

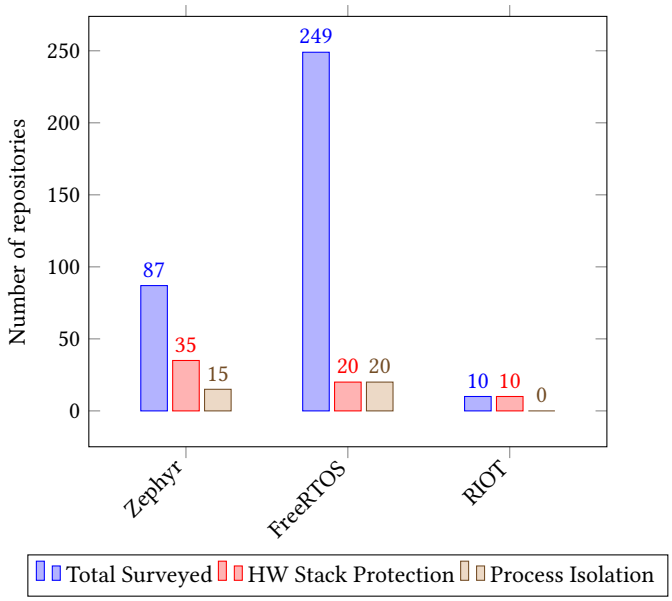

Figure 1: Surveyed embedded OS application results for enabling optional MPU features.

to a project's activity and interest from the broader community. The number of projects meeting the aforementioned criteria for each embedded OS are shown in Figure 1. Our search exclusively considers the open source projects we are able to access through GitHub's API; our survey does not account for the numerous closed source projects that also utilize the selected embedded OSes.

After curating our downstream application dataset, we now search this dataset to determine which projects enable the selected configurable security features. For the purposes of this study, we combine process–process and kernel–process isolation under the umbrella term of process isolation. We use repository specific keywords, shown in Table 3, to perform our search.

We identify three interesting survey findings below. Each selected Table 4 result provides contextualization to the developer usage of configurable security features:

Result 1: Zephyr & FreeRTOS MPU Feature Usage. Zephyr and FreeRTOS offer similar MPU-based security configurations, and both disable process isolation and stack overflow protection by default. Of surveyed Zephyr and FreeRTOS applications, 17% of Zephyr and 8% of FreeRTOS projects enable both process isolation and hardware stack guards. We subsequently observe that 89% of surveyed FreeRTOS/Zephyr projects do not enable the full suite of opt-in, configurable MPU security features. We do not include RIOT in this result as RIOT does not support MPU process isolation.

Embedded OS	Keyword	Feature
FreeRTOS	`taskCreateRestricted()`	Process Isolation
	`taskCreateRestricted()`	HW Stack Guard
	`configCHECK_FOR_STACK_OVERFLOW`	SW Stack Guard
Zephyr	`CONFIG_USERSPACE`	Process Isolation

	`CONFIG_USERSPACE`
	`CONFIG_HW_STACK_PROTECTION`
	`CONFIG_MPU_STACK_GUARD`
		HW Stack Guard

RIOT	`mpu_stack_guard`	HW Stack Guard

Table 3: Embedded OS configuration keywords used to search repository dataset and determine configurable feature usage.

	Hardware StackGuard	Process Isolation	Software Stackguard	Total Surveyed
FreeRTOS	20	20	121	249
Zephyr	35	15	–	87
RIOT	10	–	–	10

Table 4: Embedded OS application usage of configurable security features.

Result 2: SW/HW Stackguard Usage. Surveyed FreeRTOS repositories exhibit a more than sixfold increase of software-based stackguards usage in comparison to hardware-based stack guard usage.

Result 3: Opt-out Configuration Usage. For RIOT platforms possessing an MPU, RIOT enables `mpu_stack_guard` by default. We find that no surveyed project disables this feature.

3.2 Benchmarking

We further investigate the overhead of enabling hardware-based stack protection. We use an nRF52840DK board featuring an ARM Cortex-M4F CPU and execute ten million context switches with `CONFIG_USERSPACE` enabled and disabled respectively. We count yield-operations between two threads and record the overall time overhead of enabling `CONFIG_USERSPACE`. We measure the overall time in ticks.

Result 4: Runtime Overhead. We find that the average overhead from ARM MPU-based memory protection is 224 CPU cycles per context switch.

4 Case Study

We select two widely used projects as case studies of real world embedded security. These projects were chosen for their popularity and inclusion in many consumer devices.

4.1 Zephyr OS – Embedded Controller Firmware

Intel provides a Zephyr OS-based reference firmware for *Embedded Controllers*.[3] The Embedded Controller (EC) is a dedicated microcontroller present in x86 platforms. It is responsible for performing many low-level system tasks and has a high degree of control over the host system. For instance, an EC is responsible for power sequencing, handling keyboard and mouse input, thermal management, and other tasks. As such, an EC represents a privileged central system component that runs a significant number of independent tasks and interfaces with many different peripherals.

The provided reference firmware by Intel targets, among other EC chips, the Microchip MEC172x controller which features an ARM Cortex-M4F with support for the MPU. This firmware is organized as multiple tasks (such as for power sequencing and thermal management) where each task accesses a set of dedicated and shared peripherals, for instance the common Enhanced Serial Peripheral Interface (eSPI) bus connected to the host CPU.

This reference firmware does not enable any of Zephyr's optional isolation or security mechanisms. We identified code-paths where functions called from within the context of one task directly operate on memory managed by another task (e.g., power sequencing and the keyboard input controller). This code pattern is incompatible with hardware-based process isolation, which would require all interactions between tasks to go through kernel-mediated channels, such as Inter-Process Communication. As a consequence, downstream users of this reference firmware cannot easily enable additional isolation features without substantially changing the firmware's architecture.

4.2 FreeRTOS – InfiniTime

InfiniTime[4] is a firmware for the PineTime smartwatch. This firmware is built on the FreeRTOS embedded OS and the NimBLE Bluetooth Low Energy library. This firmware supports running multiple apps, an Apple HealthKit integration, and over-the-air updating. The PineTime watch is built using the nRF52832, a Cortex-M4 featuring an MPU.

InfiniTime uses software-based stack guards and exclusively uses privileged tasks with shared global state. Because privileged tasks are able to access all peripherals and memory, their usage in PineTime allows any tasks to interfere with another tasks. To test this claim, we use the InfiniSim InfiniTime simulator. InfiniTime presents a system display task and a heartbeat monitor task that share access to a common object containing references to all other apps. To demonstrate that task state is shared, we invoke `app->heartRateController.Stop()` within the display refresh loop and are able to stop the heart rate controller in InfiniSim.

5 Discussion of Results

Our results show that a majority of surveyed projects do not explicitly enable optional, off-by-default OS security features (**Result 1**). We now discuss potential reasons for this trend.

[3]https://github.com/intel/ecfw-zephyr For this case study, we consider revision `dd258906a7db9f` of May 14, 2024 of this repository.
[4]https://github.com/InfiniTimeOrg/InfiniTime. For this case study, we consider revision f8f8993fac0bdd of June 19, 2024.

5.1 Device vs. Developer Overhead

Hardware security features unavoidably increase a system's runtime and potentially space overhead as they require the OS to store MPU related state and re-configure MPU registers upon context switches. FreeRTOS and Zephyr both explicitly state this concern in their documentation. Zephyr's recommended security practices explain that:

> *[memory protection] is optional as it leverages hardware features (such as Memory Protection Units or Memory Management Units), and incurs in some overhead that smaller systems might not be able to afford.* [7]

FreeRTOS echoes this sentiment in MPU documentation stating: "*If you need every last drop of performance out of your processor, then the overhead of using an MPU might be a deal breaker for you*" [6].

Our preliminary benchmarking of Zephyr in **Result 4** finds that hardware security features impose a mere 224 cycle overhead per context switch. This result agrees with the above documentation excerpts and shows an overhead that is in fact noticeable, but likely only problematic for high performance applications. We argue that the vast majority of IoT embedded applications do not require "every last drop of performance," but instead that the *primary overhead impeding the usage of configurable security features is not performance but developer overhead.*

Result 2 supports this hypothesis, with a sixfold increase in software-based stack guard usage compared to hardware stack guards in FreeRTOS.[5] These hardware stack guards are instantiated through the use of "restricted tasks" which in turn are implemented with the help of the MPU. Using restricted tasks prevents state from being shared across processes. Conversely, software stack guards also incur overhead, however they do not require manual MPU configuration. Instead, software stack guards need only be enabled during compilation and are entirely hidden from the developer. We attribute this to be a major contributing factor towards the prevalence of software- over hardware-based stack guards. In short, this suggests that features with little development overhead are more likely to be used.

Our case studies highlight the challenge of enabling hardware protection mechanisms as both analyzed applications utilize shared state across processes. Because shared memory is disallowed[6] across isolated processes, enabling hardware stack guards and process isolation for these applications would require re-architecting their codebase. This would require integrating interprocess communication and increase development complexity. This complexity is detailed in FreeRTOS' documentation, stating that:

> *Use of an MPU necessarily makes application design more complex because: first the MPU's memory region restrictions must be determined and described to the RTOS; and second the MPU restricts what application tasks can and cannot do.* [2]
>
> *You should also be wary that working with an MPU can be difficult and, at times, frustrating. It will take*

[5]FreeRTOS hardware stack guards are enabled via restricted tasks, which also confer process isolation and prevent shared state across tasks.
[6]Some MPU configurations allow tasks to be placed within the same memory pool.

> *more time to design your application as you must consider MPU regions for each of your tasks. Mistakes in these regions, such as incorrect region lengths, permissions or not linking the data of your application correctly can be confusing to debug.* [6]

6 Closing Thoughts

This paper shows the disparity between the configurable security features the studied embedded OSes offer and the features that are actually used *in the wild*. Our preliminary results and analysis suggest the primary overhead limiting the use of these features is developer overhead. Nonetheless, further investigation is required to gain a more complete understanding of why these features are left disabled by developers. So long as downstream applications see limited use in enabling such optional features, the potential security benefits embedded OSes can provide is left unrealized. This is particularly problematic given the increased attack surface of modern IoT devices.

As is, this survey provides a sampling of popular OSes, but is somewhat limited in scope. To address this, future work can expand this study to include OSes such as Mbed OS, NuttX, LiteOS, and ChibiOS/RT. Beyond understanding why developers do not enable opt-in hardware security features, investigating why OS designers allow these features to be optional in the first place warrants further study. Our preliminary RIOT results suggest that opt-out features are less likely to be disabled. Although this seems to imply that opt-out security features are superior, there likely exists a design tradeoff. Subsequently, better understanding the motivations for opt-in security can provide useful context for designing improved interfaces and modular design.

References

[1] Amazon Web Services, Inc. 2024. FreeRTOS. https://www.freertos.org/. Accessed: 2024-07-03.

[2] Amazon Web Services, Inc. 2024. *Memory Protection Unit (MPU) Support.* Accessed: 2024-06-27.

[3] Amazon Web Services, Inc. 2024. Stack Usage and Stack Overflow Checking. https://www.freertos.org/Stacks-and-stack-overflow-checking.html. Accessed: 2024-07-03.

[4] ARM Limited. 2024. TrustZone for Cortex-M: System-Wide Security for IoT Devices. https://www.arm.com/technologies/trustzone-for-cortex-m. Accessed: 2024-07-08.

[5] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 234–251. https://doi.org/10.1145/3132747.3132786

[6] Adam Lewis. 2024. Benefits of Using the Memory Protection Unit. https://www.freertos.org/2021/02/benefits-of-using-the-memory-protection-unit.html. Accessed: 2024-07-03.

[7] Anas Nashif. 2024. Best practices for Secure Zephyr Applications and Deployment. https://github.com/zephyrproject-rtos/zephyr/wiki/Security-Best-Practices. Accessed: 2024-07-03.

[8] George Oikonomou, Simon Duquennoy, Atis Elsts, Joakim Eriksson, Yasuyuki Tanaka, and Nicolas Tsiftes. 2022. The Contiki-NG open source operating system for next generation IoT devices. *SoftwareX* 18 (2022), 101089. https://doi.org/10.1016/j.softx.2022.101089

[9] RIOT Project Contributors. 2024. RIOT. https://www.riot-os.org/. Accessed: 2024-07-03.

[10] WITTENSTEIN high integrity systems Ltd. 2024. SafeRTOS. https://www.highintegritysystems.com/safertos/. Accessed: 2024-07-01.

[11] Zephyr Project Contributors. 2024. Zephyr Memory Protection Design. https://docs.zephyrproject.org/latest/kernel/usermode/memory_domain.html. Accessed: 2024-07-03.

[12] Zephyr Project Contributors. 2024. The Zephyr Project. https://www.zephyrproject.org/. Accessed: 2024-07-03.

[13] Wei Zhou, Zhouqi Jiang, and Le Guan. 2023. Understanding MPU Usage in Microcontroller-based Systems in the Wild. In *Workshop on Binary Analysis Research* (San Diego, CA, USA) *(BAR '23)*. https://doi.org/10.14722/bar.2023.23007